Functional Programming and Haskell

Jonas Hübotter

February 13, 2022

1 Exercises

1.1 Haskell warm up

1. [1, p. 46] Explain the effect of the function defined here:

```
mystery :: Integer -> Integer -> Integer -> Bool
mystery m n p = not ((m==n) && (n==p))
```

   Hint: if you find it difficult to answer this question directly, try to see what the function does on some example outputs.

2. [1, p. 47] Define a function

```
threeDifferent :: Integer -> Integer -> Integer -> Bool
```

   so that the result of `threeDifferent m n p` is `True` only if all three of the numbers `m`, `n` and `p` are different.

   What is your answer for `threeDifferent 3 4 3`? Explain why you get the answer that you do.

1.2 Recursion, pattern matching

1. Define a recursive function `pow :: Integer -> Integer -> Integer` such that `pow x n` $= x^n$.

2. Define a recursive function

```
ascending :: Ord a => [a] -> Bool
```

   such that `ascending xs = True` if `xs` is in ascending order.

3. Define a recursive function

```
zip' :: [a] -> [b] -> [(a,b)]
```

   that mirrors the behavior of the Prelude function `zip`.

4. Define a function `insertionSort :: Ord a => [a] -> [a]` that sorts the input using the insertion sort algorithm.

5. [sheet 15] Decide for each of the following functions whether they are tail recursive:

   (a)
```
prod :: Num a => a -> [a] -> a
prod n [] = n
prod n (m:ms) = prod (n*m) ms
```

(b)
```
prod :: Num a => [a] -> a
prod [] = 1
prod (m:ms) = if m = 0 then 0 else m * prod ms
```

(c)
```
prod :: Num a => a -> [a] -> a
prod n [] = n
prod n (m:ms) = if m == 0 then 0 else prod (n*m) ms
```

6. [emdterm 2013] Give a *tail recursive* implementation of the following function:

```
fac :: Int -> Int
fac n | n > 0 = n * fac (n - 1)
      | otherwise = 1
```

7. [1, p. 85] *(everyone loves geometry)* Suppose we want to find out the maximum number of pieces we can get by making a given number of straight-line cuts across a piece of paper. With no cuts we get one piece; what about the general case?

   Define the function

```
regions :: Integer -> Integer
```

   such that using n straight lines `regions n` returns the maximum number of regions a two dimensional space can be divided into.

8. [retake 2013] Implement the function `remdups :: Eq a => [a] -> [a]` that removes all duplicates from a list. The first occurrence of every element should remain. For example, `remdups [1,5,3,1,0,3] == [1,5,3,0]`.

   The functions `nub` and `nubBy` from the Prelude may not be used.

1.3 List comprehensions

1. Define the function `concat' :: [[a]] -> [a]` using list comprehensions that given a list of lists, joins each element of the outer list and returns a new list.

2. Define a function `primes :: Int -> [Int]` using list comprehensions that returns all primes up to n.

   Hint: think about what auxiliary functions you could define that might help.

3. [1, p. 115] Define the function

```
matches :: Integer -> [Integer] -> [Integer]
```

   which picks out all occurrences of an integer n in a list using list comprehensions. For instance,

```
matches 1 [1,2,1,4,5,1] = [1,1,1]
matches 1 [2,3,4,6]     = []
```

   Using `matches` or otherwise, define a function

```
elem' :: Integer -> [Integer] -> Bool
```

   which is `True` if the `Integer` is an element of the list, and `False` otherwise. For the examples above, we have

```
elem' 1 [1,2,1,4,5,1] = True
elem' 1 [2,3,4,6]     = False
```

1.4 QuickCheck

1. Write QuickCheck tests to check your implementation of `zip'`, and `concat'` against the Prelude functions `zip`, and `concat` respectively.

2. [endterm 2015] Write for the given function `occs` one or more QuickCheck tests that form a *complete* test suite. A test suite is complete if every test succeeds for any correct implementation and if for any incorrect implementation at least one test fails for appropriate test parameters.

   The function `occs :: Eq a => [a] -> [(a, Int)]` counts how often each element occurs in a list. The returned list must be sorted in decreasing order by the number of occurrences and may not contain any duplicates. Elements with the same number of occurrences may be returned in any order. Elements that do not occur in the input are not allowed to be present in the output.

   Examples for correct behavior:

   ```
   occs "mississippi" ==
     [('i', 4), ('s', 4), ('p', 2), ('m', 1)]
   occs "mississippi" ==
     [('s',4), ('i', 4), ('p', 2), ('m', 1)]
   ```

   Examples for incorrect behavior:

   ```
   occs "mississippi" ==
     [('s', 4), ('i', 4), ('m', 1), ('p', 2)]
   occs "mississippi" ==
     [('s', 4), ('i', 4), ('m', 1), ('p', 2), ('x', 0)]
   occs "mississippi" ==
     [('s', 4), ('s', 4), ('i', 4), ('p', 2), ('m', 1)]
   ```

   Important: It is not required to implement the function `occs`.

1.5 Higher-order functions

1. Define the function `takeWhile' :: (a -> Bool) -> [a] -> [a]` similar to the Prelude function `takeWhile` that takes a function f and a list $[x_1, \ldots, x_n]$ and returns $[x_1, \ldots, x_{k-1}]$ where $f(x_k)$ is the first element that is false.

2. [sheet 6] Write a function `iter :: Int -> (a -> a) -> a -> a` that takes a number n, a function f, and a value x and applies f n-times with initial value x, that is `iter n f x` computes $f^n(x)$. A negative input for n should have the same effect as passing $n = 0$. For example, `iter 3 sq 2 = 256`, where `sq x = x * x`.

3. [sheet 6] Use `iter` to implement the following functions *without* recursion:

   (a) Exponentiation: `pow' :: Int -> Int -> Int` such that `pow' n k` $= n^k$ for all $k \geq 0$.

   (b) The function `drop' :: Int -> [a] -> [a]` similar to the Prelude function `drop` that takes a number k and a list $[x_1, \ldots, x_n]$ and returns $[x_{k+1}, \ldots, x_n]$. You can assume that $k \leq n$.

   (c) The function `replicate' :: Int -> a -> [a]` similar to the Prelude function `replicate` that takes a number $n \geq 0$ and a value x and returns the list containing the element x n-times.

4. Implement the left-associative fold `foldl' :: (a -> b -> a) -> a -> [b] -> a` similar to the Prelude function `foldl`. Then sketch the evaluation of

   ```
   foldl (+) 0 [1,2,3]
   ```

5. Using `foldr`, implement the following functions:

(a) `map’ :: (a -> b) -> [a] -> [b]` similar to the Prelude function `map` such that map’$(f, [x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)]$.

(b) `filter’ :: (a -> Bool) -> [a] -> [a]` similar to the Prelude function `filter` such that filter’$(f, [x_1, \ldots, x_n]) = [x_i \mid f(x_i)]$.

6. [sheet 7] Using `foldr`, implement the following functions:

   (a) `compose :: [(a -> a)] -> a -> a` such that compose$([f_1, \ldots, f_n], x) = f_1(\ldots(f_n(x))\ldots)$.

   (b) `fib :: Integer -> Integer` computing the Fibonacci numbers.

1.6 Lambda abstractions

1. [endterm 2013] Which of the following function definitions define the same function, and which do not? Justify briefly.

```
f1 xs x = filter (> x) xs
f2 xs = \x -> filter (> x)
f3 = \xs x -> filter (> x) xs
f4 x = filter (> x)
```

2. [endterm 2015] For both expressions, give a λ-expression with the same semantics that does not contain a section:

   (a) `(++ [1])`

   (b) `(++) [1]`

1.7 Review

1. [endterm 2020] Write a function `halfEven :: [Int] -> [Int] -> [Int]` that takes two lists xs and ys as input. The function should compute the pairwise sums of the elements of xs and ys, i.e. for $xs = [x_0.x_1\ldots]$ and $ys = [y_0, y_1, \ldots]$ it computes $[x_0 + y_0, x_1 + y_1, \ldots]$. Then, if $x_i + y_i$ is even, the sum is halved. Otherwise, the sum is removed from the list. An invocation of `halfEven` could look as follows:

```
halfEven [1,2,3,4] [5,3,1] = [3,2]
halfEven [1] [1,2,3] = [1]
```

   Implement the function in three different ways:

   (a) As a list comprehension without using any higher-order functions or recursion.

   (b) As a recursive function with the help of pattern matching. You are not allowed to use list comprehensions or higher-order functions.

   (c) Use higher-order functions (e.g. map, filter, etc.) but no recursion or list comprehensions.

2 Homework

2.1 Recursion, pattern matching

1. [1, p. 86] Using the addition function over the natural numbers, give a recursive definition of multiplication of natural numbers.
2. [1, p. 86] The integer square root of a positive integer n is the largest integer whose square is less than or equal to n. For instance, the integer square roots of 15 and 16 are 3 and 4 respectively. Give a primitive recursive definition of this function.
3. [sheet 1] Define a recursive function

   ```
   argMax :: (Integer -> Integer) -> Integer -> Integer
   ```

   such that `argMax g n` maximizes `g` in the domain $\{0, \ldots, n\}$.
4. [sheet 2] Define a function `toSet :: [Integer] -> [Integer]` using list comprehensions such that `toSet xs` removes all duplicates of `xs`.
5. [sheet 3] Implement a Haskell function `mergeSort :: [Integer] -> [Integer]` that sorts an integer list in ascending order by using Merge Sort.
6. [sheet 15] Consider the function `concat :: [[a]] -> [a]` that concatenates a list of lists:

   ```
   concat [[1,2],[],[5,6],[7]] = [1,2,5,6,7]
   ```

   Give a tail recursive implementation of `concat`.
7. [endterm 2014] Give a *tail recursive* implementation of the function `sum :: [Integer] -> Integer` that calculates the sum of the element of the given list. With the exception of primitive arithmetic operations you may not use any other predefined function.

2.2 List comprehensions

1. Define a function `quicksort :: Ord a => [a] -> [a]` that sorts the input using the Quicksort algorithm.
2. [1, p. 115] Give a function

   ```
   duplicate :: String -> Integer -> String
   ```

   which takes a string and an integer, n. The result is n copies of the string joined together. If n is less than or equal to 0, the result should be the empty string, `""`, and if n is 1, the result will be the string itself. Use list comprehensions.
3. [1, p. 424] Give a function

   ```
   perms :: Eq a => [a] -> [[a]]
   ```

   which returns a list of all permutations of a given list.

   Hint: you may use the list difference operator `(\\) :: Eq a -> [a] -> [a] -> [a]` where xs \\ ys removes each element of ys from xs.

2.3 QuickCheck

1. Write QuickCheck tests to check your implementation of `elem'` against the Prelude function `elem`.

2. [sheet 2] Define a function `isSet :: [Integer] -> Bool` such that `isSet xs` holds iff `xs` is a set. Then check that `isSet $ toSet xs` holds using QuickCheck.

3. [sheet 1] Let `g` be the following function

```
g :: Integer -> Integer
g n = if n < 10 then n*n else n
```

   Examine `g` to determine when `argMax g n` $\neq$ `n`. Use your observations to write a function `argMaxG :: Integer -> Integer` that does not use `g` and satisfies the property `argMax g n = argMaxG n`. Write a QuickCheck test to check the equivalence.

4. [sheet 15] Write one or more QuickCheck tests for the function `sortP` as defined below. The tests should be complete, i.e. every correct implementation of `sortP` passes every test and for every incorrect implementation there is at least one test that fails for suitable test parameters.

   The function `sortP :: (Ord a, Eq b) => [(a,b)] -> [(a,b)]` sorts a list of tuples with regard to the first element of the tuple in ascending order. Tuples with the same first element may occur in any order.

   Examples for correct behavior:

```
sortP [(3, 'a'), (1, 'b'), (2, 'c')] = [(1, 'b'), (2, 'c'), (3, 'a')]
sortP [(3, 'a'), (1, 'b'), (3, 'c')] = [(1, 'b'), (3, 'c'), (3, 'a')]
sortP [(3, 'a'), (1, 'b'), (3, 'c')] = [(1, 'b'), (3, 'a'), (3, 'c')]
```

   Examples for incorrect behavior:

```
sortP [(3, 'a'), (1, 'b'), (2, 'c')] = [(1, 'a'), (2, 'b'), (3, 'c')]
sortP [(3, 'a'), (1, 'b'), (3, 'c')] = [(1, 'b'), (3, 'a'), (3, 'a')]
sortP [(3, 'a'), (1, 'b'), (2, 'c')] = [(3, 'a'), (2, 'c'), (1, 'b')]
```

   Important: It is not required to implement the function `sortP`.

5. [endterm 2013] Given the type `Nat` of natural numbers ($\{0, 1, 2, \dots\}$) and a function `stutt :: [Nat] -> [Nat]` that maps $[n_1, \dots, n_k]$ to

$$[\underbrace{n_1, \dots, n_1}_{n_1\text{-times}}, \dots, \underbrace{n_k, \dots, n_k}_{n_k\text{-times}}]$$

   Example: `stutt [2,0,3,1] == [2,2,3,3,3,1]`.

   Give a *complete* test suite consisting of two of the following QuickCheck tests:

```
prop_stutt_length ns = length (stutt ns) == sum ns
prop_stutt_contents ns = all (> 0) ns ==> nub (stutt ns) == nub ns
prop_stutt_null = stutt [] == []
prop_stutt_single n = stutt [n] == replicate n n
prop_stutt_cons n ns = stutt (n : ns) == replicate n n ++ stutt ns
prop_stutt_reverse ns = reverse (stutt ns) == stutt (reverse ns)
prop_stutt_distr ms ns = stutt ms ++ stutt ns == stutt (ms ++ ns)
```

   Justify your answer briefly. For the function `replicate :: Nat -> a -> [a]` `replicate` $m\ x = [\underbrace{x, \dots, x}_{m\text{-times}}]$ holds.

6. [retake 2014] In this task sets are represented by lists. As usual, duplicates and ordering do not have any effect on the represented set. For example, the lists `[1, 2]`, `[1, 2, 1]`, and `[2, 2, 1]` all represent the mathematical set $\{1, 2\}$.

   Consider the function `takeAny :: [a] -> (a, [a])`. The input xs is interpreted as set A. The function chooses a random element a of A and returns a pair (a, ys). ys is a list representing the set $A \setminus \{a\}$.

   Is the set A empty, the function may behave in any way (e.g. not terminate).

   Examples:

```
takeAny []                          -- not defined, random
takeAny [1] = (1 ,[])
takeAny [1 ,1 ,1] = (1 ,[])
takeAny [1 ,2] = (1 ,[2]) or (2 ,[1]) or (1 ,[2 ,2]) or ...
takeAny [1 ,2 ,2] = (1 ,[2]) or (2 ,[1]) or (1 ,[2 ,2]) or ...
takeAny [1 ,2 ,3] = (1 ,[2 ,3]) or (3 ,[2 ,1 ,2]) or ...
```

   As the examples show, the function is allowed to create duplicates or change the ordering.

   Give a *correct* and *complete* QuickCheck test suite for this function. Justify your answer briefly.

7. [retake 2013] Given a function `h :: [Float] -> [Float]` that maps a list $[x_1, \ldots, x_n]$ of numbers to the list $[\frac{1}{x_1}, \ldots, \frac{1}{x_n}]$. The behavior of the function for inputs that contain the number 0 is not specified.

   Example: `h [2.0, 4.0, 0.5] == [0.5, 0.25, 2.0]`.

   Give a *complete* test suite of at most two QuickCheck tests describing the behavior of `h`.

2.4 Higher-order functions

1. Define the function `dropWhile' :: (a -> Bool) -> [a] -> [a]` similar to the Prelude function `dropWhile` that takes a function f and a list $[x_1, \ldots, x_n]$ and returns $[x_k, \ldots, x_n]$ where $f(x_k)$ is the first element that is false.

2. [sheet 7] Write a function `iterWhile :: (a -> a -> Bool) -> (a -> a) -> a -> a` such that `iterWhile test f x` iterates `f` until `test x (f x)` is false, and then returns `x`.

3. [sheet 7] Use `iterWhile` to implement a function `fixpoint :: Eq a => (a -> a) -> a -> a` that iterates a function `f` until it finds a value `x` such that `f x = x` and then returns this value.

4. [sheet 7] Use `iterWhile` to implement a function `findSup :: Ord a => (a -> a) -> a -> a -> a` such that `findSup f m x` finds the larges value $f^n x$ that is at most m assuming that f is strictly monotonically increasing.

5. [sheet 7] Using `foldr`, implement the following functions:

   (a) `length' :: [a] -> Integer` computing the length of a list.
   (b) `reverse' :: [a] -> [a]` reversing a list.
   (c) `inits' :: [a] -> [[a]]` computing the prefixes of a list.

6. [1, p. 252] Define a function

```
slope :: (Float -> Float) -> (Float -> Float)
```

   which takes a function f as argument, and returns (an approximation to) its derivative f' as result.

7. [1, p. 252] Define a function

```
integrate :: (Float -> Float) -> (Float -> Float -> Float)
```

   which takes a function f as argument, and returns (an approximation to) the two argument function which gives the area under its graph between two end points as its result.

2.5 Lambda abstractions

1. [endterm 2015] Give an expression describing the same function as

```
\xs ys -> reverse xs ++ ys
```

You may not use λ-expressions or define auxiliary functions. The use of function composition `(.) :: (a -> b) -> (b -> c) -> (a -> c)` is permitted.

2.6 Review

1. [retake 2014] Define a function `lups :: Ord a => [a] -> [a]` that returns the longest, strictly monotonously increasing (uninterrupted) partial list of the input.

   Examples:

```
lups [] = []
lups [2 ,2 ,1] = [2] or [1]
lups [2 ,5 ,3 ,6 ,8] = [3 ,6 ,8]
lups [3 ,7 ,2 ,8] = [3 ,7] or [2 ,8]
```

2. Implement the functions `zip' :: [a] -> [b] -> [(a,b)]` and `unzip' :: [(a,b)] -> ([a], [b])` similar to the Prelude functions `zip` and `unzip` in three different ways:
   (a) As a recursive function with the help of pattern matching. You are not allowed to use list comprehensions or higher-order functions.
   (b) As a list comprehension without using any higher-order functions or recursion.
   (c) Use higher-order functions (e.g. map, filter, etc.) but no recursion or list comprehensions.

3. [endterm 2013] Consider the function `f :: [Int] -> [Int]` that maps a list xs onto the list of absolute values of the negative numbers in xs.

   Example: `[1,-2,3,-4,-5,6]` should be mapped to `[2,4,5]`.

   Implement f in three different ways:
   (a) As a recursive function; without the use of list comprehensions or higher-order functions.
   (b) With the help of a list comprehension; without the use of recursion or higher-order functions.
   (c) With the help of `map` and `filter`; without the use of list comprehensions or recursion.

References

[1] Simon Thompson. *Haskell - the craft of functional programming.* 3rd ed. Pearson, 2011.